Tutoriat SO 8: Filesytem Implementation

December 28, 2025

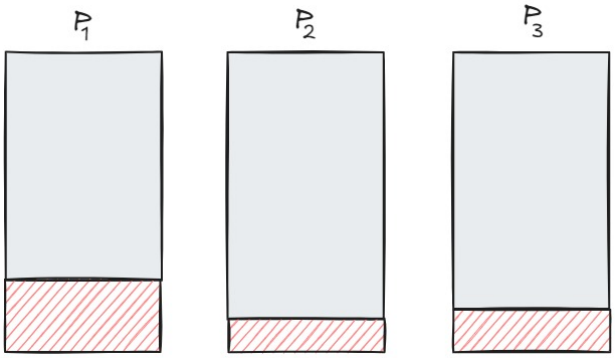
Contents

1 Memory fragmentation

1.1 Internal fragmentation

Some memory management techniques rely on dividing the main memory into fixed-size blocks and allocating those blocks to processes/files. Obviously, there is no wasted memory between the blocks, but there can be wasted space **within** the blocks.

Internal fragmentation occurs when an allocated memory block of size X is larger than the actual memory needed of size Y; a waste of $X - Y$ arises.



1.2 External fragmentation

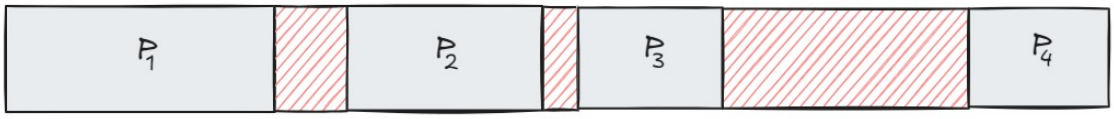
When free memory is available in the system, but it is divided into many small, scattered chunks, it can be impossible to satisfy a request for a single, large, contiguous block of memory, even though the *total* amount of free memory could be sufficient. This is known as **external fragmentation**.

For example, imagine allocating some processes blocks of **variable** sizes. When a process finishes its execution and is removed from memory, it leaves behind a **hole** of free memory. Since the allocated blocks are of variable sizes, the holes are also of variable sizes.

When a new process of size X arrives, it is possible that there may be no space for it to be stored contiguously in memory. Even if there is an infinite amount of holes of size Y, they are useless if $Y < X$.

When searching for a hole to accommodate a process, three main strategies can be applied:

- **First fit:** allocate the first hole that is big enough. Searching can either start at the beginning or at the location where the previous first-fit search ended.
- **Best fit:** allocate the smallest hole that is big enough. The whole list must be searched, unless it is ordered by size.
- **Worst fit:** allocate the largest hole.



2 Filesystem Structure

Filesystems have layered structure like the following:

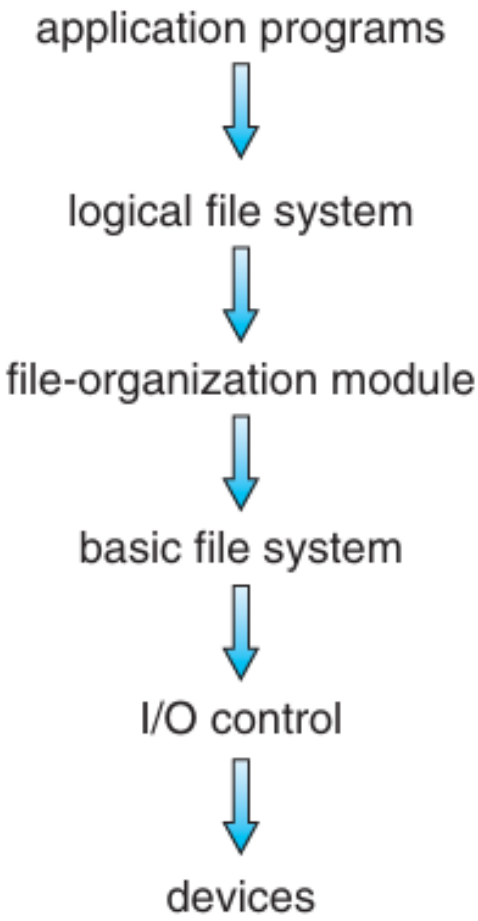


Figure 1: Filesystem layers

Each level in the design uses the features of lower levels to create new features for use by higher levels.

I/O Control:

Device drivers and interrupt handlers which get input such as "retrieve block 123" and it's output are hardware specific instructions that will become input for the I/O con-

troller(hardware) which does the actual disk head movement.

Basic Filesystem:

- Translates logical block addresses to physical block addresses that are then passed to the I/O control layer.
    - Handles scheduling for I/O requests.
    - Handles memory buffers and caches.

File Organization Module

- The file-organization module knows about files and their logical blocks. Each file's logical blocks are numbered from 0 (or 1) through N.
    - Converts the per-file logical block numbers to filesystem level logical block numbers.
    - Also handles communication with the free-space manager.

Logical Filesystem

- Handles metadata information about the filesystem(directories, files, anything but the actual data of the files)
    - The metadata of the files is stored in file control blocks(FCB, also called inodes), holding information such as ownership, permissions and location of the file contents

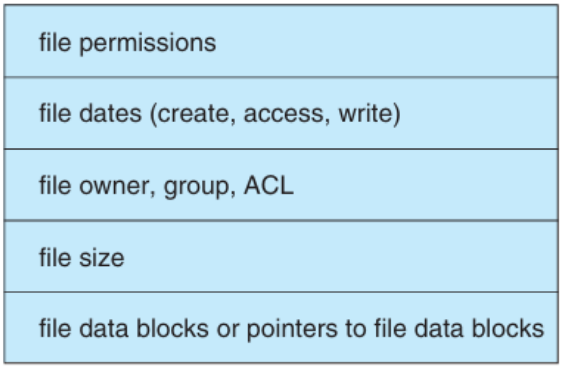


Figure 2: FCB structure

3 Filesystem Operations

Here are some important structures needed for filesystem/system operations that reside on persistent storage(disk):

- Boot control block (per volume)can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume.

- A volume control block(per volume)contains volume details,such as the number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers.

- A directory structure (per file system) is used to organize the files and contains pairs of filename - inode.

Note: Unix pairs filenames with inode numbers. The inode number is used as an index in an inode table which actually holds the inodes.

And the following are in-memory:

- system wide open file table: Keeps copies of FCBs of open files.

- per-process open file table: Keeps pointers to entries in the system wide table only for the process that owns it.

- cache for recently accessed directories

Usage

File Creation

When a file gets created, a new FCB is created(or picked from a list of free FCBs, if everything is allocated at system startup), a new entry is added to the directory structure (in memory) and then saved to disk.

File open/close

- When a file is first opened, an entry in the system wide table is created. Each global entry also has a reference count.

-All opens will create an entry in the per-process table and will point to that global entry, incrementing it's reference count.

- All closes will remove an entry from the per-process table, decrementing it's reference count. The file gets closed only when the reference count is 0.

4 Directory implementation

4.1 Linear list

The directory is implemented as a list, where each element contains the necessary metadata to define a file. The elements contain pointers to the data blocks (where the actual file content is stored).

To create a new file, a linear search must be performed through the entire list to ensure that no existing file has the same name; if this condition is met, a a new entry is added at the end of the directory.

To delete a file, a linear search is performed and the space allocated to it is released. A few ways of reusing a deleted entry could be as follows:

- Mark it as unused by assigning it a special name (such as an all-blank name, or by including a used-unused bit in each entry).
- Attach it to a list of free directory entries.

- Copy the last entry in the directory into the freed location and decrease the length of the directory.

The main disadvantage of this approach is the linear search, which is slow. Binary searches could speed the process up if the list were sorted, but maintaining a sorted list can potentially be computationally expensive. A more sophisticated data structure (such as a type of balanced trees) can help in this situation.

4.2 Hash Table

A linear list stores the directory entries, but they are accessed with the help of a **hash function**. A hash table is used, that takes a hash value computed from the file name and returns a pointer to the entry in the linear list.

Directory search time is greatly decreased; insertion and deletion are significantly improved. However, collisions must be resolved. Additionally, a major difficulty would be the fixed size of the hash table and the dependence of the hash function on that size. For a hash table that holds 64 entries, attempting to create a 65th file would require a size readjustment, which results in the need to update the hash function to match the new size.

Alternatively, a chained-overflow hash table can be used. Each entry is a linked list instead of an individual value, and collisions are resolved by adding the new entry to the linked list. Search time can be slow due to possibly large linked lists, but it is still better than performing a linear search on the whole directory.

5 Allocation Methods

5.1 Contiguous Allocation

The file is stored contiguously (it's blocks are consecutive). The directory entry keeps track of the address of the first block and number of blocks.

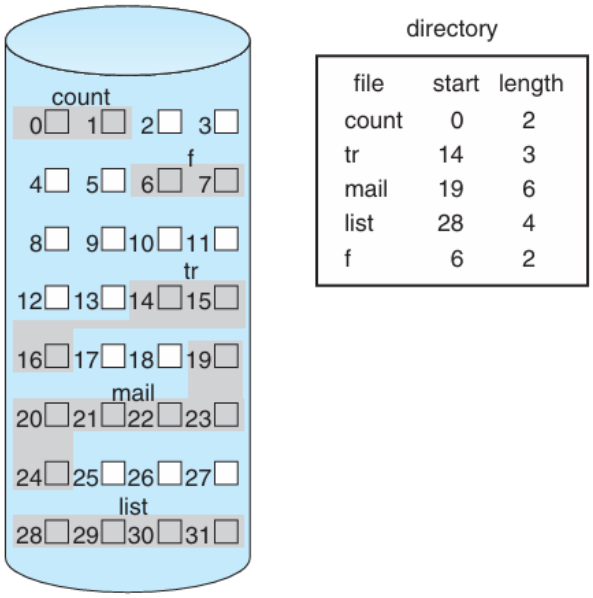


Figure 3: Contiguous allocation example

This technique is simple and supports both sequential (I can "iterate" over the file's blocks really fast) and direct access (I can access a random block of the file really fast).

But it comes with a lot of problems:

1. Finding space for a new file

In our earlier example from the image, assume another file comes in consisting of 8 blocks. There is no area of 8 contiguous block available so we can't store the file even though clearly we have more than 6 blocks available.

This problem is created because of **external fragmentation**, and one way of fixing it to periodically copy all the file from one disk to another in a contiguous manner, having all files one after another, but that can be very costly and might require down time which is might not be acceptable.

2. Dynamic changes in file size

Another file comes and now has a size of 6 blocks. We can fit it in the zone starting from block 8 and ending at block 13. Now the user opens that file and starts writing in it. This means the file is growing in size which means we need another block. But the next block, 14 is already used so we can't use it.

3. Space pre-allocation for files

Sometimes we know the size of the file (like for the earlier examples, maybe a file copy/move), but maybe we just created a new file.

If we allocate too little we might end up with problem 2 from earlier. If we allocate too much we might end-up wasting storage.

5.1.1 Modern Solution

A contiguous chunk of space is allocated initially. Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an extent, is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

5.2 Linked Allocation

5.2.1 Classic

A file's blocks are stored randomly in storage, each having a pointer to the next block in the sequence. The directory entry keeps the pointer to the first and last block.

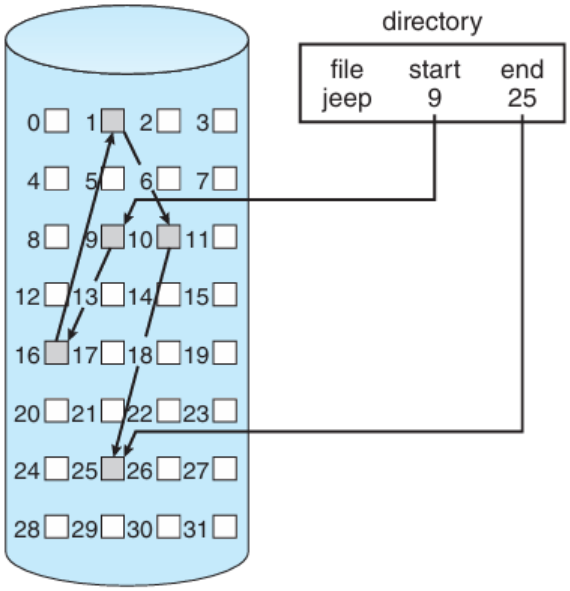


Figure 4: Linked allocation example

Even if this solution solves all the earlier problems it still has it's disadvantages:

Overhead:

Each block holds the "next" pointer of each block, which costs 4 bytes (it differs based on the system actually).

This problem can be solved using clusters, which means blocks are grouped, so instead of keeping one pointer per block we keep one pointer per group.

This solution has other advantages such as less head seeks, easier logical to physical block mappings and decrease in the amount of space needed for storage management.

But it can also worsen **internal fragmentation** and be inefficient for small data request.

Reliability

With everything connected, breaking one thing breaks everything after it.

Looking at the image, imagine that the pointer from block 1 to 10 gets corrupted somehow, this means we can't access anything starting from block 1 onward.

5.2.2 FAT

A variation of the linked list technique presented is to keep all the pointers in one place called the file allocation table(FAT).

- The index is a block number, the value is the number of the next block in the sequence.
- Unused blocks have a value of 0 and end of file blocks have a special non-zero value.
- The directory keeps the number of the first block

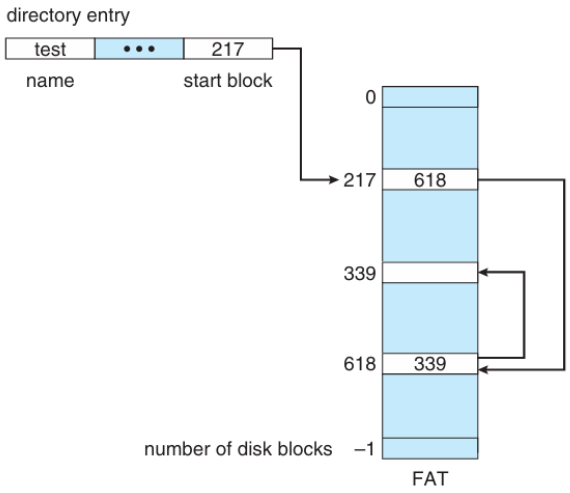


Figure 5: FAT example

This technique improves random access as less disk-head movement is required to find a given block(having everything in the same area).

But with every access we first need to look at the fat then jump to the actual block which brings overhead.

Note: The FAT itself is also stored on the disk

5.3 Indexed Allocation

Besides the actual blocks containing the data of the file, an index block also exists, per file, which holds an array of all the blocks representing the file (i-th position gives the address of the i-th block in the file's block sequence). The directory entry keeps a pointer to that index block.

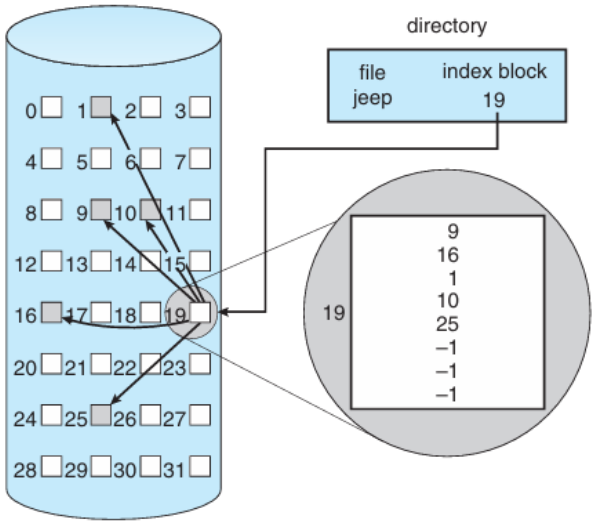


Figure 6: Index block allocation example

This might end up wasting more memory that the linked list approach, as we might be using a whole block to store only a few pointers. For this problem here are a few approaches:

Linked Scheme

At the end of the index node's array is a pointer to another index node. Thus, for big files we have a linked list of index nodes, each having arrays of pointers to actual data blocks.

Multilevel Scheme

Instead of keeping pointers to blocks, an index node can keep pointers to other index nodes which in turn have pointers to actual blocks. So you end up with an array of pointers to arrays of pointers to blocks. This can be extended to multiple such levels, not just 2

With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

Combined scheme

We could have multiple levels in the same index node. For example some of the pointers can be direct accesses to blocks. Some (less than earlier) could be 1-level indirect accesses. Some (less then earlier) could be 2-level indirect accesses. And so on.

Unix uses this technique, 12 slots are for direct accesses, 1 for 1-level indirect, 1 for 2-level and 1 for 3-level. All this data, together with other file metadata is stored in what's called an inode. All inodes exist in a table which lives on a preallocated zone on the disk.

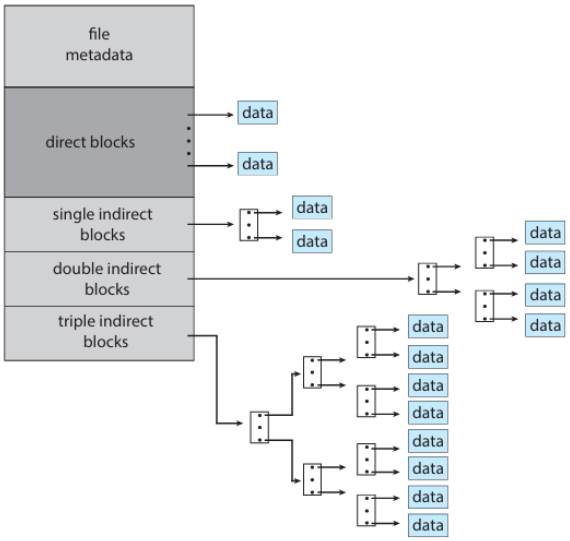


Figure 7: Combined scheme

6 Free Space Management

Here are a few approaches to keeping track of available disk blocks:

6.1 Bit Vector

A vector of bits where the value of the i'th index tells us whether block i is allocated(0) or not(1). Here is an example where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest allocated:

00111100111111000110000011100000

Figure 8: Bit vector example

To get the first free block we have to find the first non-0 word(as a word can be 0 only if all the bits in the word are 0). The formula for calculating the number of the first free block is:

nrOf0Words * wordSize + offsetOfTheFirst1Bit

6.2 Linked List

Only keeps the the available blocks as a linked list, where the first node is the first available block.

6.3 Grouping

Each node in the linked list actually keeps n pointers to free blocks and a pointer to the next node in the list.

6.4 Counting

A variation of the grouping approach is for each node to contain a pointer to a free block and count of the number of contiguous blocks starting from there. This similar to the extent method for contiguous memory allocation.

7 Efficency

- Unix keeps inodes scattered around the volume, as close as possible to the actual file blocks to reduce seek time.

- BSD UNIX uses clusters with varying sizes, based on the file's size to avoid internal fragmentation.

- Solaris used fixed size process and open file tables which meant that if the size reached it's limit no more processes could be created / no more files could be opened. This fixed size allocation saved time (as everything was initialized at start-up) but provided limited functionality.

8 Recovery

- Because filesystems have structures both in memory and on disk inconsistencies can appear.

- An example would be allocating a new block, which might result in it being removed from the free list but not added in the inode's list, losing the block.

- Another example involves the use of memory buffers. If we only write to the memory buffer and flush it to the disk when it's full a crash might cause data loss.

8.1 Consistency Checking

- Scan the directory entries and actual disk blocks for inconsistencies.

- How successful the system is at recovering is based on the allocation scheme.

- The linked list approach can reconstruct the file and put it in a lost+found/ directory even if the directory entry is lost

- The index approach would create a leak, as the blocks are still alocated but the file can't be reconstructed